STATE OF ALABAMA

Information Technology Guideline

Guideline 660-01G2: Input Validation and Data Integrity

1. INTRODUCTION:

One of the major causes of software vulnerabilities is failure to validate input. Common security vulnerabilities found in applications as a result of un-validated input include:

- Buffer Overflow
- Integer Overflow
- Command Injection
- Format String
- Cross-site Scripting

This document describes these vulnerabilities, describes how to detect their presence in applications, and describes how to avoid or eliminate them.

2. OBJECTIVE:

Ensure applications perform checks on the validity of input data, user permissions, and resource existence before performing a function.

3. SCOPE:

These guidelines may be used for both in-house application development and to assist in the evaluation of the security of third-party applications. The guidance provided is not specific to any one platform, programming language or application type. Some portions of this guideline may not apply to all applications or in all cases. In some cases, specific guidance has been provided based upon platform, programming language, or language types.

4. GUIDELINES:

The following guidelines are based on the recommendations of the Defense Information Systems Agency (DISA) published in the Recommended Standard Application Security Requirements Security Technical Implementation Guide (STIG).

4.1 INPUT VALIDATION

Input may come from a user, data store, network socket, or other source. Before being used applications should validate all user input and input data from any source when it crosses a trust boundary.

When validating data the application should check for known-**good** data and reject any data **not** meeting predefined criteria. Checking for known-**bad** data, while potentially effective, may still allow an attacker to get bad data through the validation by using alternate data encoding or patterns the developers have not considered. Allowing only known-good data through may increase the results of valid data being rejected; however, this significantly lessens the possibility of invalid data being passed on to the application.

4.2 COMMON SECURITY VULNERABILITIES

The following is not an exhaustive list of vulnerabilities, nor is every vulnerability applicable to every application environment.

Managers, designers, developers, and testers should actively monitor application security developments for new classes of vulnerabilities.

4.2.1 Buffer Overflow

Buffer overflow is a situation where data is written beyond the end of an allocated memory block. There are several types of buffer overflow which all lead to the same potential issues in the application, the ability to crash the application or execute arbitrary code on the system. If the vulnerable application is running as a system account, this could lead to a compromise of the system.

The following items may indicate potential buffer overflow within the application:

- Cases where input is not checked before being copied into a buffer
- Use of some of the functions listed in Table 4-1 (below)
- Incorrect calculations to determine buffer sizes
- Incorrect calculations to determine array indexes

The primary methods of detecting buffer overflow are code reviews and Fuzz testing. Fuzz testing is the process of sending large data blocks and invalid data blocks as input to an application in an attempt to cause an application error.

In order to minimize buffer overflow implement the following procedures:

- Replace function to be avoided with safer functions (see Table 4-1 below)
- Recheck all calculations to ensure buffer sizes are calculated correctly
- Recheck all array access and flow control calculations
- (C++) Replace character arrays with STL string classes
- Validate all input before use, allowing only known-good input through

4.2.1.1 Unsafe Functions

The following C/C++ functions have a high potential for causing application vulnerabilities. The presence of these functions does not immediately indicate a vulnerability, however if they are used inappropriately a vulnerability may exist.

Table 4-1: C/C++ Functions to Avoid

Unsafe Function	Risk	Potential Replacement Functions
strcpy	Buffer Overflow	strcpy_s, StringCchCopy, StringCbCopy, strlcpy
wcscpy		
_tcscpy		
_mbscpy		
strCpy		
strCpyA		
strCpyW		
Lstrcpy		
lstrcpyA		
lstrcpyW		
strcpyA		
strcpyW		
_tccpy		
_mbccpy		
strcat	Buffer Overflow	strcat_s, StringCchCat, StringCbCat, strlcat
wcscat		
_tcscat		
_mbscat		
strCat		
strCatA		
strCatW		
lstrcat		
lstrcatA		
lstrcatW		
strCatBuffW		
strCatBuff		
strCatBuffA		
strCatChainW		
strcatA		
strcatW		
_tccat		
_mbccat		
lstrlen	Buffer Overflow	StringCbLength or StringCchLength

4.2.1.2 String Considerations

Many functions, messages, and macros use strings in their parameters. Failing to check strings for null-termination or for length or miscalculating the length of a string or buffer can lead to buffer overflow or data truncation. To handle strings in a safe manner:

- Check strings for null-termination or for the proper length, as appropriate
- Determine the length of a string or buffer, especially when it is a TCHAR
- When creating a string, or using a string that was used previously, initialize it to zero or insert a null-terminator (as appropriate)
- Use the Strsafe.h functions (Microsoft) when dealing with C/C++ strings

4.2.2 Integer Overflow

Integer overflow results from inconsistent and/or incorrect results from calculations due to the handling of integer data types in various programming languages. These errors can lead to symptoms ranging from crashes and incorrect results to allowing an attacker to exploit a buffer overflow. Integer overflow most often occur because developers are not aware of the implicit conversions occurring when using variables of different data types or when using different operators.

The following may indicate the presence of integer overflow in the application:

- Mixing signed and unsigned data types in calculations or comparisons
- Mixing data types of different sizes in calculations or comparisons
- Comparisons between variables and literal values
- Use of un-validated input
- Calculations not validated before the result is utilized

The primary method of detecting integer overflows in an application is a code review. If a code review cannot be performed, test the application to ensure it correctly handles different numeric values and strings of varying lengths. The following items list some of the tests to be performed.

- Input negative values for numeric input
- Input border case values (i.e., 0, 7, 8, 254, 255, 16353, 16354)
- Input extremely large string values (> 64k)
- Input strings whose lengths equal border cases (32k, 32k-1, 64k, 64k-1)

In order to minimize integer overflow implement the following procedures:

- Use unsigned values whenever possible
- Use only unsigned integers in memory allocation whenever possible
- Use only unsigned array indexing functions whenever possible
- Validate user input of numeric value, allowing only known good data to pass
- Compile with the highest warning level possible
- (C++) Use size_t types to hold size variables
- (C++) Use the SafeInt class (available from Microsoft)
- (C#) Compile with the /checked compiler option
- (GCC) Compile with the –ftrapv option

4.2.3 Command Injection

Command Injection vulnerabilities exist when data that can be modified to execute some task is passed to an interpreter or compiler. This can allow an attacker to execute code, possibly at a higher privilege level, resulting in system compromise.

Potential Command Injection vulnerabilities include the use of functions spawning an interpreter or compiler. Table 4-2 lists some common functions vulnerable to Command Injection.

Table 4-2: Functions Vulnerable to Command Injection

Language	Functions/Characters
C/C++	system(), popen(), execlp(), execvp(), ShellExecute(), ShellExecuteEx(), _wsystem()
Perl	system, exec, `,open, |, eval, /e
Python	exec, eval, os.system, os.popen, execfile, input, compile
Java	Class.forName(), Class.newInstance(), Runtime.exec()

In addition to code reviews, testing should be performed to help identify Command Injection vulnerabilities. In order to test for Command Injection vulnerabilities:

- Identify all potential interpreters or compilers used to pass data
- Identify the characters modifying the interpreters' behavior
- Construct input strings containing these characters causing a visible effect on the system, pass them to the application, and observe the behavior (ensure all input vectors are tested in this manner)

In order to minimize Command Injection vulnerabilities, validate all input before passing to an interpreter or compiler, allowing only known-good input through.

4.2.4 Format String

Format string vulnerabilities occur when specially crafted format strings passed to a function allow flow control information to be viewed or modified or cause the program to read or write arbitrary memory locations. In a worst-case scenario format string vulnerabilities can allow an attacker to execute code of their choice on the system resulting in complete system compromise.

An application taking input and passing it to a formatting function may indicate the presence of format string vulnerabilities. The primary method of detecting this vulnerability is a source code review. If a source code review cannot be performed, then application testing may be used. To test the application insert format string specifiers in all areas of string input and observe the output looking for anomalies. The format specifiers will vary based upon the language used; however, in addition to language-specific specifiers, the C language specifiers should also be tested.

In order to minimize format string vulnerabilities, implement the following procedures:

- Validate all input before passing it to a function; allow only known-good data to pass

- Format strings used by the application should only be accessible by privileged users
- (C++) Use stream operators instead of the printf family of functions

Table 4-3: C/C++ Unsafe Functions (Printf Family)

Unsafe Function	Risk	Potential Replacement Functions
Printf	Format String Vulnerabilities	Add format string (“%s”) as a first argument
fprintf		
sprintf	Format String Vulnerabilities	snprintf
vsprintf	Format String Vulnerabilities	vsnprintf
wsprintf	Format String Vulnerabilities	StringCbPrintf, StringCbPrintfEx, StringCbVPrintf, StringCbVPrintfEx, StringCchPrintf, StringCchPrintfEx, StringCchVPrintf, StringCchVPrintfEx.
wvsprintf		

4.2.5 Cross Site Scripting (XSS)

XSS is a situation where input is accepted by a web site and then sent back to a web page. This input can include code to be executed by the browser. Since this code is seen as originating from the web server it can access data from the servers’ domain such as a cookie, or modify the behavior of the web site by modifying links and other malicious actions. A cross site scripting vulnerability can lead to an attacker gaining personal information or directing a user to a site of the attacker’s choice.

If user input is echoed back into the browser, the application may have potential XSS issue.

In addition to code reviews, testing should be performed to identify potential XSS vulnerabilities. In order to test for XSS vulnerabilities:

- Make a request against the application, setting all input parameters to known-bad values
- View the HTML response, looking for the known-bad values sent as input (the response containing the values submitted as input may not be returned immediately, it may go through intermediate processes before being sent back to a browser or may be sent in response to a future request or possibly to a different entity)

In order to minimize XSS vulnerabilities in the application, implement the following procedures:

- Filter all input, allowing only known-good input
- If special characters are required for input, HTML encode user input
- Set a known character for all web pages to eliminate unexpected characters

4.3 DATA INTEGRITY CONTROLS

4.3.1. The application should use a NIST-approved technology to implement a hash (e.g., Secure Hash Algorithm One [SHA-1]), checksum, or digital signature of the data before transmission.

4.3.2. When code-signing is required, the application should invoke a NIST-approved digital signature technology to digitally sign the code prior to transmission.

4.3.3. The application should invoke NIST-approved technology to apply a hash, checksum, or digital signature to the data before storage.

4.3.4. The application should be able to validate the integrity mechanism and reject data for which the integrity mechanism validation fails.

4.3.5. The application should validate parameters before acting on them and reject all parameters for which one or more of the following is true:

(1) Not formatted as expected;

(2) Do not fall within the expected bounds (length, numeric value, etc.)

4.3.6. The application should inform the user of the expected characteristics of the input—e.g., length, type (alphanumeric, numeric only, alpha-only, etc.), and numeric or alphabetic range.

4.3.7. The application should validate all data input by users or external processes and reject all input for which one or more of the following is true:

(1) not formatted as expected

(2) contains incorrect syntax

(3) not a valid data string

(4) contains parameters or characters with invalid values

(5) falls outside the expected bounds (e.g., length, range)

(6) contains a numeric value that would cause a routine or calculation in the application to divide any number by zero

(7) contains any parameters the source of which cannot be validated by the user's session token

(8) can induce a buffer overflow

(9) contains HTML

(10) contains special characters, meta code, or meta-characters that have not been encoded (if encoding is allowed)

(11) contains direct SQL queries

(12) contains any other type of unexpected content or invalid parameters

(13) contains a truncated pathname reference

4.3.8. The application should suspend all processing of the transaction in which input has been received until the input has been completely validated.

4.3.9. All application programs, including CGI and shell scripts, should perform input validation on arguments received before acting on those arguments.

4.3.10. The application should validate all inputs it receives from any external processes, including processes in third-party software integrated into the application, in the same way it validates user input data.

4.3.11. All functions in the application program should perform bounds checking, such that the functions check the size of all buffer or array boundaries before writing to them, or before allowing them to be written to. In addition, the application should limit the size of what it writes to the buffer or array to the size imposed by the buffer/array boundaries (i.e., to prevent what is written from exceeding the buffer/array size and overflowing the boundary).

4.3.12. The application should bounds check all arrays and buffers every time those arrays/buffers are accessed.

4.3.13. The application should validate all input data before copying those data into the database.

4.3.14. The application should reject any input containing HTML (including HTTP strings that contain HTML tags) from an untrusted user or other untrusted source.

4.3.15. The application should be able to recognize questionable URL extensions, and validate all URLs sent to it by clients to ensure they do not contain such extensions or truncate the URL to remove the dubious extension.

4.3.16. The application should not accept Web page content from any untrustworthy source. The application should verify and validate the source of any Web page content before posting that content.

4.3.17. All user input validations should be performed by the server application even if input validation has already been done by the client application. The client application should not be relied on to perform trustworthy input validation. Client input validation may be used to prescreen data before it is validated by the server.

4.3.18. The application's validation of user input data that contains active content (e.g., mobile code) should not result in the execution of the active content.

4.3.19. The application's data update processes should operate correctly, and should not incorrectly reparse, inadvertently introduce errors to, or otherwise corrupt the data they update.

4.3.20. The application should find and validate the digital signature and any hash, checksum, or other additional integrity mechanism applied to that code prior to executing it. If the code's integrity mechanism cannot be validated, or is not present, the application should discard the code without executing it; and audit this discard.

4.3.21. The application should invoke a virus scanning tool to scan all files received from users and external processes to ensure these files do not contain malicious content.

4.3.22. The application should invoke a virus scanning tool whenever it executes a program that may access one of the application's configuration or other parameter-containing files.

4.3.23. The application should ensure that the data to be forwarded do not contain or point to malicious code.

4.3.24. The process that validates the application's executable code integrity mechanism checksum or hash should be invoked every time the application is executed to ensure that the application's executable code state has not changed since the original integrity mechanism was applied. If this validation fails, the validation process should prevent the application from being executed, and notify the administrator that the application code needs to be replaced by an uncorrupted executable.

4.3.25. The application should not execute received code until it:

(1) verifies that the code has been digitally signed; and

(2) validates the digital signature on the code.

4.3.26. The application should time/date stamp each data modification or file update.

4.3.27. The application should display to each user who retrieves the data the time and date on which the data were last modified.

4.3.28. The application code should explicitly initialize all of its variables when they are declared.

4.3.29. The application should validate the source of all HTML updates to hidden fields and should reject any HTML field changes from unvalidated sources.

4.3.30. The application should not embed parameter data about fields in HTML forms in hidden fields in those HTML forms.

4.3.31. The application should use hash or digital signature to ensure the integrity of transmitted forms (user-to-server) containing sensitive information.

4.3.32. The application should not trust user input or other user-supplied data that have not been received over a trustworthy channel, unless those data are encrypted and digitally signed.

4.3.33. The application should never return sensitive information in response to input from untrustworthy sources.

4.3.34. The application cannot be used to bypass the access controls or to spoof the trusted user to modify data within database entries/records (data integrity), the relationships between those entries/records (relational integrity), or the references to those entries/records (referential integrity)

5. DEFINITIONS:

6. ADDITIONAL INFORMATION:

6.1 POLICY

Information Technology Policy 660-01: Application Security

6.2 RELATED DOCUMENTS

Information Technology Guideline 660-01G1: SQL Injection

Signed by Art Bess, Assistant Director

7. DOCUMENT HISTORY:

Version	Release Date	Comments
Original	7/14/2008	